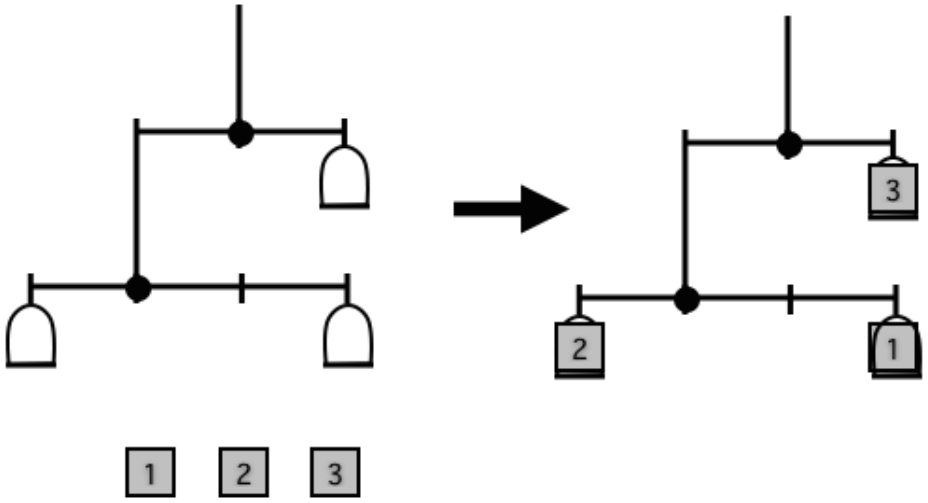
Computational Problem Solving CSCI-603
Balance Puzzles Lab 8

In a balance puzzle, you are given a structure such as the one shown below on the left, which is hanging from the ceiling, and a set of weights. Your job is then to place the weights in the pans such that each horizontal beam is balanced, as shown in the solution below on the right. For a beam to be balanced, the torque must be zero. Each item hanging from the beam provides a torque equal to its weight times the distance from the support point of the beam. We assume that the beams themselves and the vertical hangers have no weight. So, for example, in the balance shown here, the bottom beam is balanced because the left weight and right weight each provide two units of torque, and the upper beam balances because each each side provides three units. Note that the absolute length of each beam is irrelevant, we only need to know the relative distances of each hanging point from the center to determine if the torque is balanced. The tick marks along each beam are assumed to be at equal spacing.

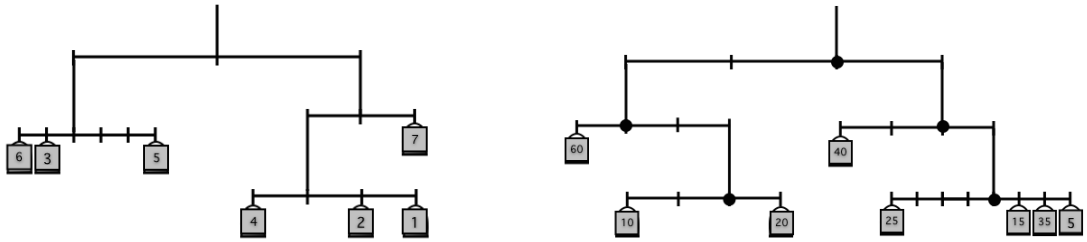


In this lab, we will not be attempting to solve balance puzzles (perhaps another day!). Instead, we will be checking them and figuring out how to draw them so that the various beams and weights do not overlap.

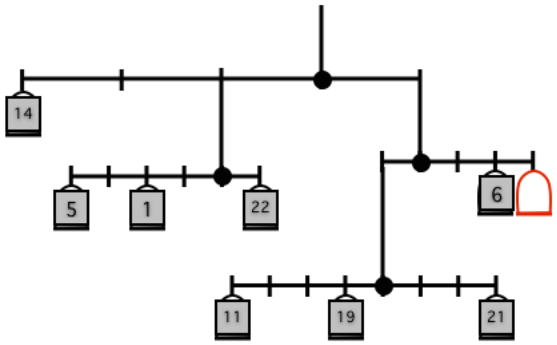
Problem Solving

Work in a team of three to four students as determined by the instructor.

1. Consider the following balances. Are they balanced? Show the torque on each beam in each balance.



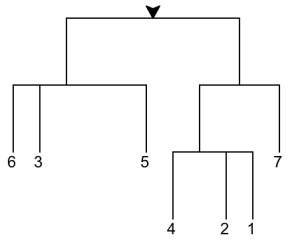
2. Consider the following balance that has one empty pan. What value weight should be placed in the pan to make it balance?



3. Consider how to represent a balance puzzle in a program. Assume that a beam will be represented by an object of the class `Beam`. What slot(s) should a Beam have? What would you expect the slot(s) to contain?

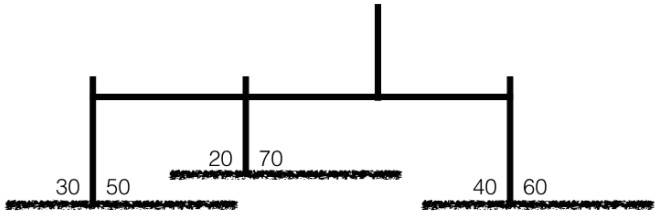
4. Based on your previous answer, assume that both `Beam`s and `Weight`s have a `weight()` `function`. Write pseudocode for `Beam.weight()`.

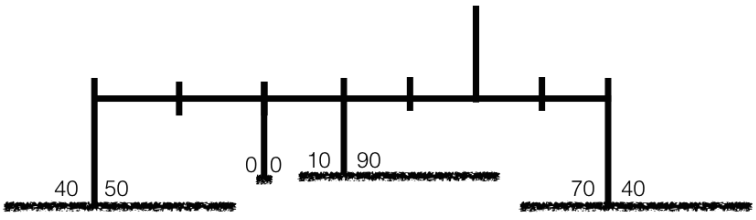
5. We would also like to draw balance puzzles from a text description. When drawn, it might look something like this:



Importantly, we do not want different beams to overlap. To make sure this happens, we will have to scale each beam left-to-right to separate the things that hang from it so that they do not overlap. In fact, we would like to leave a gap of at least 20 pixels between any two adjacent things that hang below. To simplify things, we will **not** allow any vertical overlaps such as appear in the puzzle in question 2.

In each of the following partial balance puzzles, the numbers on top of each lower beam represent the number of pixels from the center of that beam to its farthest extent (left and right). They have been drawn at different heights just for clarity here, in practice they would all be drawn at the same height. The zeros in the second balance refer to a weight, that we assume has 0 width.





(a) For each balance, what should be the scale factor (pixels per tick along the beam) so that there is at least 20 pixels of space between adjacent lower beams/weights?
(b) For each balance, how far left and right will it extend relative to its hanging point? You should include the width of those things hanging from it as well.

Implementation

You will implement two classes, `Beam` and `Weight`, and a main function that reads a balance puzzle from a file and verifies it and draws it.

A balance puzzle will be defined in a file as a list of beams, one per line of the file. Each beam will have a name followed by a list of things attached, and where they are attached:

```
<name> <dist1> <hanger1> <dist2> <hanger2> ...
```

You may assume that each name is `B` followed by a number, except for the topmost beam which is simply called `B`. All distances will be integers and all hangers will be either previously-defined beam names or positive integers representing weights. For example, the left puzzle in question 1 of problem solving will be represented as:

```
B1 -2 6 -1 3 3 5
B2 -1 4 1 2 2 1
B3 -1 B2 1 7
B -1 B1 1 B3
```

You should read this data in and construct the appropriate objects to represent it. Some puzzles will be given in which a single weight is given as -1, representing an empty pan. You can assume that there is an integer that can be put in that spot to make the puzzle balance, and your code must compute the value.

The ordering of the data file guarantees that when a beam is given, you can then determine the left-right extent that it will take up and the scale at which you will need to draw it. In particular, note that the scale factor required for each beam is based on the extents of the things that hang from it, and the extent of a beam is based on the scale factor and the extents of the things that hang from it. Weights (as in the problem solving) can be assumed to have zero left-to-right extent.

Once you have read in the entire puzzle and computed the scale for each beam, your code should open a turtle window and draw the balance puzzle. This should be implemented as a `draw` function within the `Beam` class (perhaps with a small external helper function), rather than external to the class. You can use the `turtle.write` function to write the values of the weights — you may need to play around a little with turtle motions to get the text to appear where you want it. The actual drawing should be done simply by calling `draw` on beam `B`, which will then call the `draw` function on those things hanging from it, and so on. Make sure that you get your preconditions and postconditions correct for all drawing functions!

Submission

One team member should upload all the source files to your CS account and run the following try command:

```
try grd-603 lab8-1 balances.py
```

Grading

- Problem solving: 20%
- Implementation: 75%
  - File reading: 5%
  - Balancing check: 25%
  - Finding the missing weight: 10%
  - Computing the extent of a beam: 20%
  - Drawing a balance puzzle: 15%
- Style: 5%

1 Problem solving solutions

1. Left (in postorder): 6*2+3*1 == 5*3 ; 4*1 = 2*1+1*2 ; 7*1 == 7*1; 14*1 == 14*1 ; Yes, balanced
   Right: 10*2 == 20*1 ; 60*1 == 30*2 ; 25*4 == 15*1+35*2+5*3 ; 40*2 == 80*1 ; 90*2 != 120*1 ; Not balanced
2. 51*1 == 6*2 + ?*3 ; solve for ? to get 13.
3. Probably just a dictionary of lever-arm-distance to thing hung there.
4. Something like:

```
def weight(self):
    wt = 0
    for lever in self.hangers:
        wt += self.hangers[lever].weight()
    return wt
```

5. (a) Top balance: Left side pair needs 90 pixels for one tick, right pair needs 130 for two ticks. So 90/tick is the minimum that works for both.
   Bottom balance: Pixels/ticks for each pair: 70/2 = 35 ; 30/1 = 30 ; 180/4 = 45. So 45/tick required.
   (b) Top balance: Left of center = 90*2 + 30 = 210, right = 90 + 60 = 150 Bottom balance: Left = 5 * 45 + 40 = 265, right = 2*45 + 40 = 135.